Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

Identifying functionality of fragments in concurrent programs *

Ting-Lu Huang
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsin-Chu, Taiwan
e-mail: tlhuang@csie.nctu.edu.tw

Abstract

To identify the functionality of a program fragment is to determine the effect to program behavior caused by removing the fragment. When given a concurrent program with the proof of all desired properties, we are assured the correctness. Left unsaid are often the purposes of various fragments in the program text. A scenario approach to answering questions such as What is the purpose of this variable?, What is the purpose of this busy-waiting loop?, What is the purpose of this assignment?,*and so forth, is proposed. By answering such questions, one gains a component-wise understanding of the program. The task of determining functionality of fragments is shown to be less demanding than the original proof of the complete program. The gain is a further understanding not provided by proofs. Several classical solutions to the n-process critical section problem are used for illustration.*

1 Introduction

The set of possible computation sequences for a concurrent program can be so vast that an exhaustive behavioral reasoning for program correctness is difficulty to conduct, and is often regarded as unreliable[5, p.5]. However, behavioral reasoning is an effective way to refute a claim of program property over an arbitrarily large set of possible computation sequences: it suffices to provide one refuting sequence. By proposing appropriate claims about a program(or a slightly modified version) and refuting the claims with counter examples, we seek to understand the program incrementally. We call such kind of reasoning a **scenario** approach. Although correctness proof also induces understanding, its primary purpose is to establish logical truth that the program as a whole behaves as specified. It does not explicitly identify functionality of program fragments. Scenario approach is to ask questions such as *What is the purpose of this variable?, What is the purpose of this busy-waiting loop?, What is the purpose of this assignment?,*and so forth. By answering such questions, one gains a component-wise understanding of the program. Refutations of claims and correctness proofs should be viewed as complementary approaches for program understanding.

*This work was supported by National Science Council, Republic of China, under Grant NSC85-2221-E-009-039

The approach is informal in the sense that it can be immediately applied to all concurrent programs without requiring rigorous training in axiomatic reasoning or formal proof systems. It encourages behavioral observation on the target program that most programmers found comfortable with. Axiomatic reasoning or formal methods requires a language of mathematics that is not widely known or easily used even among interested groups[16]. One demanding task in our approach still remains: it requires ingenuity in observing program behavior in order to come up with appropriate claims and find the refuting sequences. One should be reminded that there are theoretical reasons to believe that the problem of proving an arbitrary program correct is Turing-undecidable [6, p.118]. And there are doubts about the usefulness of the verification tools based on formal methods[15]. For program correctness, one should not rely solely on tools. For program understanding, one should take an even more active role and be ready to apply ingenuity when called for.

Ascertaining global invariants for concurrent programs, which is the core of a proof, poses another difficulty that is often beyond the ability of most programmers: it requires a proof of non-interference[5, p.67]. One must take all actions in all other processes into account before an invariant can be established. In contrast, scenario reasoning is more intuitive and poses less difficulties because refutation requires only one computation sequence with a more focused claim in mind. The scope of concern covers a smaller set of possible computation sequences, and the language of reasoning is that of program behavior. By encouraging small steps in exploring the target, the programmer can take charge of the reasoning task at his(or her) own pace. In contrast, correctness proof encourages a global attack at the very beginning. Its first step is often critical but difficult. Scenario approach encourages a local attack step by step, so that a programmer accumulates knowledge about the program behavior incrementally.

Scenario approach encourages question-and-answer activity. To get a rough idea about the purpose of a program fragment such as a variable, we simply remove all assignment statements to that variable and all expressions involving the variable. The behavior of the crippled version must have deficiency of some sort, as a result. One task here is to find a com-

putation sequence, called **refuting sequence**, of the crippled version leading to a violation of some desired (and proved) property. A backward flow analysis is provided to assist this task. Once we find a sequence showing the violation of a property for the crippled program, we can demonstrate how the fragment in the original program works in guarding the property by replaying the same sequence we just found. The effect is a vivid display of how the fragment is working in a positive way. Several versions of bakery algorithms are shown to have respective fragment guarding the same refuting sequence threatening mutual exclusion property. The fact that all these versions contain a design to deal with the refuting sequence leads us to believe the sequence should be documented explicitly as a sound software engineering practice. In contrast, such level of understanding regarding program fragments(or design) is difficult to achieve in a typical correctness proof.

Organization of the paper follows. Section 2 sets up the criteria for determining functionality of program fragments. Section 3, 5 and 6 each presents a detailed example of scenario reasoning, each for a non-trivial solution for the critical section problem. Section 4 illustrates a backward flow analysis to assist the search for refuting sequences. Section 7 compares scenario reasoning to related works and finally, section 8 is the conclusion.

2 Criteria for determining functionality of a fragment

A problem is usually represented by a set of desired properties of the programs(solutions). For example, mutual exclusion problem[10] is formulated as requiring the following properties:

1. mutual exclusion
2. deadlock freedom
3. fairness

The first two are well understood and are the minimum requirements. Fairness is sometime optional and demands further explanations. It means that an upper bound on the number of "overtaking" for a trying process is guaranteed. A process is said to be overtaken when another process enters critical section despite of the intention to enter made by itself. If the number of overtaking is bounded by the number of processes, the fairness is **linear waiting**. If the order of entering critical section is the same as the order of making the intention to enter known, the fairness is *First-come-first-served*(**FCFS**). Note that fairness is independent of deadlock freedom: An execution sequence of deadlock contains no process having entered critical section, therefore no process is overtaking any other. Therefore, we say that the three properties are **orthogonal** to each other. There exist properties that are not orthogonal to the three, though. Starvation freedom, which guarantees eventual entrance to critical section for a trying process, is shown[10] to hold if both deadlock freedom and FCFS hold. For simplicity, we consider only orthogonal properties of a program.

The functionality of a program fragment can involve one or more properties. Criteria for determining the **functionality** P, defined as a subset of the orthogonal properties, are given as follows.

1. Without the fragment, there exists at least one refuting sequence for each of the properties in P. We say that the fragment is **vital** for P.
2. Without the fragment, all properties not in P are preserved. We say that the fragment is **irrelevant** to properties not in P.
3. If the fragment is vital for P and irrelevant for properties not in P, we say that its functionality is for P.

A variable(or fragment) can be vital for more than one properties. For example, *number* in the bakery algorithm is vital for all three properties. In such cases, the variable(or fragment) is usually the kernel of the whole design and often attracts sufficient explanation in the original document and proof. What is often left untold is the kind of design that is not the core but is necessary for correctness of the whole algorithm. An example is the *choosing* variable in the bakery algorithm. Our approach is particularly useful to those who need to understand the whole algorithm.

3 Functionality of *choosing* in the bakery algorithm

The following is the bakery algorithm as an N-process solution to the critical section problem. The common data structures are:

```
var  choosing : array[0..n-1] of boolean;
     number   : array[0..n-1] of integer;
```

Initially, these data structures are initialized to false and 0, respectively. For convenience, we define the following notations:

- $(a, b) < (c, d)$ if $a < c$ or if $(a = c$ and $b < d)$.
- $max(a_0, ..., a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, 1, ..., n-1$, and is computed as follows.

```
function max( a[0], a[1], ..., a[n-1] );
  var j, temp: integer;
  begin
     temp := 0;
     for j := 0 to n-1 do
         if temp < a[j] then temp := a[j];
     return( temp );
  end.
```

Process P_i uses the following algorithm.

```
/*                                    */
/*       The bakery algorithm         */
/*                                    */
1  repeat
2     choosing[i] := true;
3     number[i] := max(number[0],number[1],...,
                       number[n-1]) + 1 ;
```

```
4    choosing[i] := false;
5    for j := 0 to n-1
6       do begin
7          while choosing[j] do no-op;
8          while number[j] <> 0
9            and (number[j],j) < (number[i],i)
                 do no-op;
10         end;
11   {Critical Section}
12   number[i] := 0;
13   {Remainder Section}
14 until false;
```

Numerous documents contain explanations about how this algorithm works[2, 3, 5, 7, 8, 9, 12]. However, none of them provides a clear answer to the question: what is the purpose of the array *choosing*? By scenario reasoning, we would first remove line 2, 4 and 7. The resulting program text is such that *choosing* does not exist any more.

```
/*                                          */
/*    The crippled bakery algorithm         */
/*                                          */
1  repeat
3    number[i] := max(number[0],number[1],...,
                    number[n-1]) + 1;
5    for j := 0 to n-1
6       do begin
8          while number[j] <> 0
9            and (number[j],j) < (number[i],i)
                 do no-op;
10         end;
11   {Critical Section}
12   number[i] := 0;
13   {Remainder Section}
14 until false;
```

By finding a sequence of this crippled version from the initial state to one that two processes are in the critical section simultaneously, we provide an evidence that *choosing* is vital for at least mutual exclusion.

```
/* S1:                                      */
/*  a refuting sequence showing a           */
/*  violation of mutual exclusion for the   */
/*  crippled bakery algorithm               */
/*                                          */
p0 at line 3 : check max = 0
p1 at line 3 : check max = 0
p1 at line 3 : number[1] := 1
p1 at line 8 : check number[0] = 0
p1 at line 11: enter critical section
p0 at line 3 : number[0] := 1
p0 at line 8 : check number[1] = 1
p0 at line 9 : check (number[1],1) <
               (number[0],0) is false !
p0 at line 11: enter critical section
```

By using S1 again to drive the execution of the *original* bakery algorithm, we see how *choosing* is used to avoid the kind of race condition as in the sequence. Process p1 will be forced to spin at line 7 since process p0 have not finished the assignment of line 4, yet. By the time p1 is free to go, p0 must have already been given a new value of number. Process p1 would not have seen that number[0] equals 0, and would be forced to compare the *number-id* pair with that of p0. Mutual exclusion cannot be violated in this sequence.

The race condition in the sequence is not accidental in nature, but is intrinsic in the sense that all similar algorithms using the *number-id* pair must deal with it. The following algorithm, Bakery-2[9, 20, 5], helps illustrate this point.

```
/*                                          */
/*        Bakery-2                          */
/*                                          */
1  repeat
2    number[i] := 1;
3    number[i] := max(number[0],number[1],...,
                    number[n-1]) + 1;
5    for j := 0 to n-1
6       do begin
8          while number[j] <> 0
9            and (number[j],j) < (number[i],i)
                 do no-op;
10         end;
11   {Critical Section}
12   number[i] := 0;
13   {Remainder Section}
14 until false;
```

Bakery-2 avoids the race condition by using an extra assignment statement to *number* in line 2. That assignment serves the same purpose as *choosing*, however different the appearances of the fragments may be. Such level of understanding is obscure in the assertional proofs[20] or behavioral proofs[9]. It can easily be revealed by scenario reasoning, starting from the removal of the seemingly extraneous assignment: line 2. The rest of the disclosure is similar to that for the bakery algorihtm.

According to the criteria for functionality identification, we need to show that the crippled bakery algorithm satisfies deadlock freedom and fairness. For deadlock freedom, assume all contending processes are deterred indefinitely at the entry protocol. There is a time after which no process has a zero ticket number, except those that are idle. Then among these processes, there must be one whose *number-id* pair is less than those of all other contending processes. That process can proceed to enter critical section. The proof here is indeed less complicated than that for the complete program, since there is no need to consider the role of *choosing*. For fairness, we show that the crippled bakery algorithm satisfies first-come-first-served ordering. The assignment in line 3 is regarded as the action making the intention known. Observe that the processes will enter critical section in an order consistent with their *number-id* pair. Thus, fairness is preserved. The functionality of *choosing* is for mutual exclusion.

4 Finding a refuting sequence

Although finding a sequence to refute a claim requires ingenuity, we can use some knowledge about the target program and some simple techniques in guiding the search. For example, sequence S1 for the crippled bakery algorithm in section 3 can be found by a **backward flow analysis** to exclude impossible

alternatives in early stages. The finite sequence S to be found is formulated as:

$$S \equiv S_I, e_1, s_1, e_2, s_2, \ldots, S_f$$

where S_I is the initial state, S_f the state of the violation. Assume two processes P_a and P_b, with a < b, are both in critical section. There are four mutually exclusive and exhaustive cases to be considered.

1. P_a read number[b] = 0 (line 8) before entering CS; P_b read number[a] = 0 (line 8) before entering CS.
2. P_a read number[b] $\neq$ 0 (line 8) and (number[b],b) $\not<$ (number[a],a) (line 9) before entering CS; P_b read number[a] $\neq$ 0 (line 8) and (number[a],a) $\not<$ (number[b],b) (line 9) before entering CS.
3. P_a read number[b] = 0 (line 8) before entering CS; P_b read number[a] $\neq$ 0 (line 8) and (number[a],a) $\not<$ (number[b],b) (line 9) before entering CS.
4. P_a read number[b] $\neq$ 0 (line 8) and (number[b],b) $\not<$ (number[a],a) (line 9) before entering CS; P_b read number[a] = 0 (line 8) before entering CS.

For each case, we try to obtain contradiction by some simple temporal reasoning.

1. Reject it. P_a reached line 8 before P_b finished line 3; P_b reached line 8 before P_a finished line 3. A contradiction.
2. Reject it. In sequence S, number[a] and number[b] are non-decreasing. In state S_f, P_a expects that (number[a],a) < (number[b],b) holds. But P_b expects that (number[b],b) < (number[a],a) also holds in the same state. A contradiction.
3. Reject it. P_a reached line 8 before P_b finished line 3. Therefore, P_a expects that number[b]≥number[a] holds in state S_f. However, P_b expects that (number[b],b) < (number[a],a) also holds in the same state. Since we assume a < b, that is not possible.
4. Undecided.

The remaining alternative, case 4, tells a lot about what the possible sequence should look like.

- P_b reached line 8 before P_a finished line 3 since it read number[a] = 0 (line 8) before entering CS. Therefore, number[a] ≥ number[b] holds when P_a finished line 3.
- P_a read number[b] $\neq$ 0 (line 8) and (number[b],b) $\not<$ (number[a],a) (line 9) before entering CS. Therefore, it must have found that number[a] = number[b] at that time. (Since a < b and number[a] ≥ number[b] at that time)

It becomes trivial then to obtain sequence S1 in section 3.

5 Functionality of while-loops in EM algorithm.

Eisenburg-McGuire's n-process algorithm(**EM** algorithm) is another earlier one[18] but well-known[2, p.211][19, p.30]. It is a good example to show how scenario reasoning helps us understand programs. The shared variables are follows.

```
var flag: array[0..n-1] of (idle, want_in, in_cs);
    turn: 0..n-1;
```

All the elements of `flag` are initially `idle`, the initial value of `turn` is immaterial (between 0 and n-1). The structure of process P_i is:

```
/*                              */
/*       EM algorithm           */
/*                              */
   0  var j: 0..n;
   1  repeat
   2   repeat
A: 3     flag[i] := want_in;
A: 4     j := turn;
A: 5     while j <> i
A: 6        do  if flag[j] <> idle then j := turn
A: 7            else j := ( j + 1 ) mod n;
B: 8    flag[i] := in_cs;
B: 9    j := 0;
B: 10   while (j < n) and (j = i or flag[j] <> in_cs)
B: 11                    do j := j + 1;
   12  until (j >= n) and (turn = i or flag[turn] = idle);
   13  turn := i;
   14  {critical section}
   15  j := (turn + 1) mod n;
   16  while (flag[j] = idle) do (j := j + 1) mod n;
   17  turn := j;
   18  flag[i] := idle;
   19  { remainder section }
   20 until false;
```

No assertional proof can be found in literature for this algorithm. The only documented proof[18] is behavioral. In it, however, no explanation is given as regard to the purpose of fragment A from line 3 to line 7. The question arises since apparently there are two while loops a process must go through before entering its critical section. The first while loop in fragment A is not meant to guarantee mutual exclusion, for there exists the second while loop in fragment B whose purpose is understood to guarantee mutual exclusion. What is the purpose of the first while loop, then?

To confirm that the first while loop is not for mutual exclusion, we need to come up with a sequence.

```
/* S2:                                                    */
/*   sequence for EM algorithm to show simultaneous       */
/*   execution of line 8.  Initially, turn = 0,           */
/*   flag[0]=idle, flag[1]=idle.                          */
/*                                                        */
p1 at 3:  flag[1] := want_in
p1 at 4:  j := 0
p1 at 5:  check j <> 1, must do line 6
p1 at 6:  check flag[j]=idle, must do line 7
p1 at 7:  j :=  1
```

```
p1 at 5:  check j = 1, can go to line 8
p0 at 3:  flag[0] := want_in
p0 at 4:  j := 0
p0 at 5:  check j = 0, can go to line 8
/* p0 and p1 executes line 8 simultaneously */
```

The resulting state is that process p0 and process p1 can execute fragment B simultaneously.

The purpose of fragment B is clearly for mutual exclusion: before entering line 12, a process must scan the entire flag to make sure that no other process is *in_cs*. We suspect that the purpose of fragment A is for deadlock freedom. To confirm it, first we need to provide a sequence leading to a deadlock for the crippled EM algorithm which lacks fragment A.

```
/*                                              */
/* crippled EM algorithm :                      */
/*           EM with fragment A removed.        */
/*                                              */
   0  var j: 0..n;
   1  repeat
   2   repeat
B: 8     flag[i] := in_cs;
B: 9     j := 0;
B: 10    while (j < n) and (j = i or flag[j] <> in_cs)
B: 11                    do j := j + 1;
   12  until (j >= n) and (turn = i or flag[turn] = idle);
   13  turn := i;
   14  {critical section}
   15  j := (turn + 1) mod n;
   16  while (flag[j] = idle) do (j := j + 1) mod n;
   17  turn := j;
   18  flag[i] := idle;
   19  { remainder section }
   20 until false;
```

The following sequence shows that the crippled-EM suffers from deadlock.

```
/* S3:                                          */
/*     sequence for the crippled-EM             */
/*                                              */
p0 at 8:           flag[0] := in_cs
p1 at 8:           flag[1] := in_cs
p0 at 9,10,11,12:  cannot enter line 13
                   since j is less than n.
p1 at 9,10,11,12:  cannot enter line 13
                   since j is less than n.
p0 at 8:           flag[0] := in_cs
p1 at 8:           flag[1] := in_cs
/*  p0 and p1 is in deadlock */
```

To illustrate how the original algorithm with fragment A guarantees deadlock freedom, we learn from sequence S2 and sequence S3 in forming S4 in which deadlock would not happen.

```
/* S4:                                                  */
/*    sequence for EM algorithm to illustrate           */
/*    deadlock freedom. Initially, turn = 0,            */
/*    flag[0]=idle, flag[1]=idle                        */
/*                                                      */
p1 at 3:  flag[1] := want_in
p1 at 4:  j := 0
p1 at 5:  check j <> 1, must do line 6
p1 at 6:  check flag[j]=idle, must do line 7
p1 at 7:  j := 1
p1 at 5:  check j = 1, can go to line 8
p0 at 3:  flag[0] := want_in
p0 at 4:  j := 0
p0 at 5:  check j = 0, can go to line 8
/* p0 and p1 executes line 8 simultaneously   */
p0 at 8:           flag[0] := in_cs
p1 at 8:           flag[1] := in_cs
p0 at 9,10,11,12:  cannot enter line 13
                   since j is less than n.
p1 at 9,10,11,12:  cannot enter line 13
                   since j is less than n.
p1 at 3:  flag[1] := want_in
p1 at 4:  j := 0
p1 at 5:  check j <> 1, must do line 6
p1 at 6:  check flag[j]=in_cs, therefore j := 0
p1 at 5:  check j = 0, must spin until p0
          finishes line 18
p0 at 3:  flag[0] := want_in
p0 at 4:  j := 0
p0 at 5:  check j = 0, can go to line 8
/* Deadlock cannot happen since p1 will be  */
/* spinning at the first while loop.        */
```

We see from the replay of the sequence that, this time, only process p0 can enter fragment B. Therefore, the deadlock pattern can no longer continue. Process p0 can go on to enter critical section since process p1 is kept spinning at the first while loop.

According to the criteria for functionality identification, we need to prove that the crippled-EM satisfies both mutual exclusion and fairness. Mutual exclusion for the crippled-EM is trivial and is omitted. We will prove that a contending processes cannot be overtaken for more than $n-1$ times once it declared its intention to enter critical section. This is assured since *turn* is changed only by a process having access to critical section: at line 13 and at line 17. The value assigned to *turn* is always in the cyclic order of current (turn+1, turn+2, ..., n-1, 0, 1, ..., turn). The crippled-EM may suffer from deadlock, but a contending process cannot be overtaken by more than $n-1$ processes since its intention to enter critical section will be detected by a releasing process at line 16. Fairness is preserved. The functionality of fragment A is deadlock freedom.

5.1 Scenario and assertional reasoning for deadlock freedom of EM

We show how scenario and assertional reasoning can work together for proving deadlock freedom of EM. The argument in the original paper[18], basically an assertional one, is presented as follows. If none of the contending processes has yet passes line 12(the last controlling point before critical section), then after a point in time, the value of *turn* will be constant(since no statement before line 12 assigns value to *turn*). Such hypothetical scenario cannot hold up because the **first contending process** in the cyclic ordering (turn, turn+1, ..., n-1, 0, 1, ..., turn-1) *will meet no resistance* in enter critical section.

From the argument, we are confirmed deadlock freedom. But there is no explicit indication as to what fragments that are responsible for holding up

the property. Furthermore, the last sentence in its argument requires a trace of the repeat-until loop before one can conclude that indeed the first contending process will meet no resistance. In fact, to really understand that sentence, we need to know the possibility of simultaneously entering line 8, and the fact that fragment A can prevent further mutual blocking in the second run of the repeat-until loop. For we still need to explain why the first contending process will pass line 10 with no other process *in_cs*. So, in the end, the same sequence(S4) needs to be consulted before an understanding is achieved.

To enhance the original proof of EM, we continue the assertional proof of deadlock freedom as follows. Note that we still need behavioral arguments occasionally, and we benefit from the result of scenario reasoning.

```
/*    EM algorithm  with auxiliary variables      */
/*    for Owicki-Gries' style of reasoning        */
/*    Initially, run = 0.                         */
/*                                                */
   0  var j: 0..n;
   1  repeat
   2   repeat
A: 3      flag[i] := want_in;
   **     run[i] := run[i] + 1; /*auxiliary action*/
A: 4      j := turn;
A: 5      while j <> i
A: 6         do  if flag[j] <> idle then j := turn
A: 7             else j := ( j + 1 ) mod n;
B: 8     flag[i] := in_cs;
B: 9     j := 0;
B: 10    while (j < n) and (j = i or flag[j] <> in_cs)
B: 11                      do j := j + 1;
   12  until (j >= n) and (turn = i or flag[turn] = idle);
   **  run[i] := 0;          /*auxiliary action*/
   13  turn := i;
   14  {critical section}
   15  j := (turn + 1) mod n;
   16  while (flag[j] = idle) do (j := j + 1) mod n;
   17  turn := j;
   18  flag[i] := idle;
   19  { remainder section }
   20 until false;
```

The following invariant holds for process P_i in fragment B.

$$\forall\, k \neq i \;::\; (i - \text{turn}) \bmod n < (k - \text{turn}) \bmod n \;\wedge\; \text{run[i]} \geq 2 \;\wedge\; \text{run[k]} \geq 2 \;\Longrightarrow\; \text{flag[k]} = \text{want\_in}$$

To prove it, we need to establish: (1) it holds when P_i steps in fragment B, and (2) as long as process P_i stays in fragment B, no actions in any other process can change the invariant from truth to falsity. From scenario reasoning as in S4, we know claim (1) is true. The only action that modifies flag[k] are those at line 8 and line 18. But, according to our scenario reasoning, no process can pass the first while loop (which lies before line 8) except the first contending process. All those elements flag[k], as the invariant states, will remain *want_in*.

We then use the invariant to argue that, in the hypothetical deadlock that all the contending processes are forced to loop indefinitely in the repeat-until loop, the first contending process will find the invariant true at line 10. It will meet no resistance in letting j to reach n since all other process will be either *idle* or *want_in*. A process which is closer to turn may turn from *idle* to *want_in*. That process becomes the first contending process, by definition. And the same argument still holds for it. The first contending process will eventually pass the second while loop. It will meet no resistance in pass line 12 since by definition of first contending process, either P_i is *turn* or process P_{turn} is *idle*. In either case, P_i will pass line 12. Deadlock freedom is proved.

6 Functionality of *turn* in Peterson's algorithm.

Like many n-process solutions to the critical section problem, the first step in understanding Peterson's is critical and difficult. Existing documents[21, 22] provides reasoning for understanding this algorithm in particular. Scenario reasoning provides the first step in understanding all similar algorithms in general, however humble the step may be.

The shared variables are follows.

```
var   q[1..n]      :   array of integer;  /* initially 0
      turn[1..n-1] :   array of integer;  /* initially 0
```

The structure for process P_i is follows.

```
/*                                  */
/*    Peterson's algorithm          */
/*                                  */
line 1  repeat
line 2    for j := 1 to n-1 do
line 3        begin
line 4            q[i] := j;
line 5            turn[j] := i;
line 6            for k := 1 to i-1, i+1 to n do
line 7                if ( (q[k] >= j) and (turn[j]=i) )
                         go to line 6;
line 8        end;
line 9    critical section
line 10   q[i] := 0;
line 11 until false;
```

We speculate from experience that array q is for at least mutual exclusion since it is set before it is scanned in the entry protocol. One complication is that array q is set and scanned in many stages. The other is that array q is not boolean but integer. It is not a good choice for removal since the remaining would be too terse to reason about. We will remove *turn* instead to start the scenario reasoning.

When a process is executing lines 2-8 at iteration j, we say it is in the jth-stage. A crippled version is obtained by removing *turn* and a sequence is given to show that *turn* is vital for deadlock freedom.

```
/*                                          */
/*    Crippled  Perterson's algorithm       */
/*                                          */
```

```
line 1  repeat
line 2    for j := 1 to n-1 do
line 3        begin
line 4            q[i] := j;
line 5
line 6            for k := 1 to i-1, i+1 to n do
line 7                if  (q[k] >= j) go to line 6;
line 8        end;
line 9    critical section
line 10   q[i] := 0;
line 11 until false;
```

```
/* S5:                                                  /*
/*   sequence for crippled Peterson's algorithm         /*
/*                                                      /*
P1 at line 2:    j := 1
P2 at line 2:    j := 1
P1 at line 4:    q[1] := 1
P2 at line 4:    q[2] := 1
P1 at line 6:    check k=2
P1 at line 7:    q[2] >= j, go to line 6
P2 at line 6:    check k=1
P2 at line 7:    q[1] >= j, go to line 6
/* P1 and P2 are in deadlock  */
```

We show that the crippled version satisfies mutual exclusion. Assume P_a and P_b are both in critical section. Consider the cases when each of them reached the $(n-1)$-th stage respectively. When P_a reached that stage, it found that $q[b] < n-1$ was true. We say P_a finished line 4 for that stage (event A) happened before P_b finished line 4 for that stage(event B). Likewise, when P_b reached that stage, it found that $q[a] < n-1$ was true. We say P_b finished line 4 for that stage (event B) happened before P_a finished line 4 for that stage(event A). A contradiction.

We show that the crippled version satisfies linear waiting. Assume a process is eager in executing line 4 at every stage. After line 4 is executed, all other processes at lower stages are deterred until the process executes line 10. The maximum number of processes at higher stages at that time must be less than $n-1$ for which the process must wait. Linear waiting is proved.

The functionality of *turn* is for deadlock freedom.

7 Related works

It has been no secret that concurrent algorithms can be very subtle and error prone. Almost all of the algorithms have a proof of some sort in the first place. Simply put, we don't understand concurrent algorithms well. To foster better understand, functionality of important program fragments should be identified and documented. As our illustration showed, there exist many neglected opportunities for better understanding. To the best of our knowledge, no previous studies in concurrent algorithms both address this issue and illustrate it so vividly without requiring painstaking formalism. This is not to say that no previous studies attempt to explain the purpose of a particular component of a program, only that either the purpose is suggested with only words or one has to extract relevant assertions mentioned somewhere in the proof. The closest study to ours is the compositional approach proposed in[10, 13, 14]. However, they emphasize on composing independent programs into a larger one that satisfies all properties of the constituents. It is by no means a straightforward task. Our approach seems to be working on the opposite direction: to find out the functionality of program fragments.

The criteria of functionality for program fragments is a new formulation. Semantics of program constructs, like while loops and assignments, have been studied intensively[5, 1, 4]. The interpretation is at the level of precondition and postcondition on program state transition. Our formulation is more intuitive in that it links the fragment directly with the desired property set. The cause and effect is displayed at behavioral level, not at the state transition level which requires higher degree of mathematical precision.

In scenario reasoning, the refuting sequence is an evidence to turn down the claims: it will withstand rigorous examinations of any level since an error is always an error. Therefore, vitality of a fragment, which is based solely on refutation, is not subject to any doubt. Irrelevance of a fragment, however, is subject to future scrutiny if the original proof of the whole program is later found doubtful. This is because irrelevance is established by the same proof techniques as used in the original proof. Unfortunately, a proof is not always a good proof, as well illustrated by Lamport [12] regarding his earlier proof[7] for the bakery algorithm. To summarize, vitality of a fragment to a property is for sure. Irrelevance of a fragment to a property is only as good as the original proof, on which this paper does not aim to improve.

8 Conclusion

Scenario reasoning is an approach for *a posteriori* understanding of concurrent programs at the component level. It is not meant to replace proofs for understanding programs, but to provide the first step in hope that programmers would gain more and more insight to the original design, starting from identifying functionality of fragments. In contrast, a proof requires a critical global observation that is too difficult for most programmers. The proofs of the presented examples have been widely studied but are still regarded as difficult for most. When scenario reasoning was used to explain the algorithms to programmers in a classroom setting, the enthusiasm enticed can be felt. Its preliminary success is mostly due to the intuitive nature of the question-and-answer endeavor. We understand at least the cause and effect with fragment removal/installation being the cause, and the change of program behavior being the effect.

References

[1] Krzysztof R. Apt and Ernst-Rudiger Olderog. Verification of sequential and concurrent programs Springer-Verlag, New York, 1991.

[2] A. Silberschatz and P. B. Galvin. Operating System Concepts. Fourth Edition, Addison-Wesley, 1995.

[3] Alan Burns and Geoff Davies. Concurrent Programming. Addison-Wesley,1993.

[4] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3): 225–262, September 1993.

[5] G. R. Andrews. Concurrent Programming: Principles and Practice. Benjamin Cummings, 1991.

[6] Jacques Loeckx and Kurt Sieber. The Foundations of Program Verification. Wiley-Teubner, 1984.

[7] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8): 453–455, August 1974.

[8] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2): 125–143, Sep. 1977.

[9] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1): 84–97, July 1979.

[10] Leslie Lamport. The mutual exclusion problem – Part I and II. *Journal of the ACM*, 33(2): 313–348, April 1986.

[11] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1): 1–11, Feb. 1987.

[12] Leslie Lamport. win and sin : Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3): 396–428, July 1990.

[13] E. A. Lycklama and V. Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems*, 13(4): 558–576, Oct. 1991.

[14] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1): 73–132, Jan. 1993.

[15] Hossein Saiedian. An invitation to formal methods. *IEEE Computer*, 29(4): 16–32, April 1996.

[16] Kate Finney. Mathematical notation in formal specification: too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2): 158–159, Feb. 1996.

[17] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5): 279–285, May 1976.

[18] M. A Eisenberg and M. R. McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11): 999, Nov. 1972.

[19] M. Raynal. Algorithms for Mutual Exclusion. MIT Press, 1986.

[20] E.C.R. Hehner and R.K. Shyamasundar. An implementation of P and V. *Information Processing Letters*, 12(4): 196–198, Aug. 1981.

[21] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3): 115–116, June 1981.

[22] M. Hofri. Proof of a mutual exclusion algorithm. *Operating Systems Review*, 24(1): 18–22, Jan. 1990.